CS477 Takehome Final, Due at 4pm on Tuesday 5/11

Note: Should hand by the above deadline a hardcopy in latex, or similar formatting language, to Ms. Donna Coleman, my administrative assistant, who is in office SC 2106. The time of 4pm is a *hard* deadline, since she will leave her office at 4:30 sharp.

Make sure that printouts of all code and interactions with tools are included in the hardcopy. In addition to including such printouts in the hard copy, you should also email the corresponding working code by the same deadline to `meseguer@cs.uiuc.edu`. The Maude specifications and all auxiliary files or tools needed for the exercises can be retrieved from the course web page.

For any questions/problems with any of the tools involved in the homework, Ralf Sasse (`rsasse@uiuc.edu`) will be willing to help you, provided you make such requests at reasonable working hours and with sufficient anticipation.

1. Consider the following specifications involving sorting as described in Lecture 12 which you can download from the course web page:

```
fmod INT-LIST is protecting INT .
sorts List .
op nil : -> List [ctor] .
op _:_ : Int List -> List [ctor] .
endfm

fmod FRAME-SORTING-REQUIREMENTS is protecting INT-LIST .
sort Multiset .
subsort Int < Multiset .
op sorted : List -> Bool .
op null : -> Multiset .
op __ : Multiset Multiset -> Multiset [assoc comm id: null] .
op mset : List -> Multiset .
var L : List .
vars N M : Int .
eq sorted(nil) = true .
eq sorted(N : nil) = true .
ceq sorted(N : M : L) = sorted(M : L) if (N <= M) = true .
ceq sorted(N : M : L) = false if N <= M = false .
eq mset(nil) = null .
eq mset(N : L) = N mset(L) .
endfm

fmod INSERT-SORT is
protecting INT-LIST .
op ins : Int List -> List .
op sort : List -> List .
vars N M : Int .
var L : List .
eq ins(N, nil) = N : nil .
ceq ins(N, M : L) = N : M : L if N <= M = true .
ceq ins(N, M : L) = M : ins(N, L) if N <= M = false .
eq sort(nil) = nil .
```

```
eq sort(N : L) = ins(N, sort(L)) .
endfm

fmod INSERT-SORT-VERIFICATION is
protecting INSERT-SORT .
protecting FRAME-SORTING-REQUIREMENTS .
endfm
```

Use the version of the Maude ITP available on the course web page (*not* the one for the Java+ITP tool) to prove the following goals about the above sorting specification (you can also download the goals from the course web page. **Note:** See comments on the course web page about special requirements for using the ITP for the first goal):

```
select ITP-TOOL .
loop init-itp .

(goal ins1 : INSERT-SORT-VERIFICATION
      |- A{L:List}((sorted(sort(L:List))) = (true)) .)


(goal ins2 :
  INSERT-SORT-VERIFICATION |-
    A{L:List}((mset(sort(L))) = (mset(L))) .)
```

Please, include *both* the proof as a list of commands to the ITP, and a printout of your interaction with the ITP in your hardcopy.

2. To solve the following exercise use the *special* version (with support for this Java subset) of the ITP downloadable from the course web site that has an extension of the list of commands of the ITP specifically designed to support Hoare logic reasoning in our Java subset. You already used this version in Homework 6.

   The way this extension works is as follows:

   (a) start the appropriate version of Maude compatible with this tool

   (b) load into Maude the functional module defining the semantics of our language, `java-es-flat.maude`, (for speed reason we use the flattened version).

   (c) define a program (`BlockStatements`) `foo` in a module `foo.maude` importing `java-es-flat.maude` by declaring a constant `foo` and giving an equation defining the constant as the corresponding program text. Also declare a constant `foo-init` which contains all declarations necessary for your program and make sure there are no ("Java") declarations in your program `foo`. The module `foo.maude` should also contain all *auxiliary functions* needed in a proof of correctness of program `foo`. (Note that this has already been done for you, in the file maxnml.maude.)

   (d) load the following `maxnml.maude` module

```
fmod MAXNML-JAVA is
including JAVAX .
op maxValue : Int Int -> Int [assoc comm] .
vars N M : Int .
ceq maxValue(N, M) = M
    if N <= M = true .
ceq maxValue(N, M) = N
    if N <= M = false .

ops maxnml-init maxnml : -> BlockStatements .
eq maxnml-init = (int 'N ; int 'M ; int 'L ; int 'Z ;) .
eq maxnml = if ('N <= 'M)
```

```
                    (if ('M <= 'L)
                      ('Z = 'L ;)
                     else ('Z = 'M) ;)
                else (if ('N <= 'L)
                       ('Z = 'L ;)
                      else ('Z = 'N ;)) .
    endfm
```

(e) then load the Java+ITP version of `itp-tool.maude`

(f) then, to prove a Hoare triple

$$\{P\}\ \texttt{foo}\ \{Q\}$$

you give a `javax` command, or a `javax-inv` command which also needs an invariant.

In this exercise this command has already been spelled out for you in the following file "maxnml.itp",

```
select ITP-TOOL .
loop init-itp .

(javax MAXNML-JAVA :
--- specification constants
(M:Int ; N:Int ; L:Int)
--- precondition
((S:WrappedState['N]) = (int(N:Int))
 & (S:WrappedState['M]) = (int(M:Int))
 & (S:WrappedState['L]) = (int(L:Int)))
--- program
maxnml-init
maxnml
--- postcondition
((S:WrappedState['Z]) =
 (int(maxValue(N:Int, (maxValue(M:Int, L:Int))))))
.)
```

where javax is used as there is no loop in the program. Then you can interact with the ITP tool to discharge the created first-order goals.

Note that, if you wish, you can write your solution (i.e. the commands you used to discharge a goal) directly below the javax command and then in one shot load the module which will create the goal and discharge it.

If you have any problems using the Java+ITP tool, you can ask help from Ralf Sasse (`rsasse@uiuc.edu`).

3. Consider a communication channel in which messages can get out of order. There is a sender and a receiver. The sender is sending a sequence of data items, for example numbers. The receiver is supposed to get the sequence in the exact same order in which they were in the sender's sequence. To achieve this in-order communication in spite of the unordered nature of the channel, the sender sends each data item in a message together with a sequence number; and the receiver sends back an ack indicating that has received the item. The specification in Maude of the protocol is as follows (you can retrieve it from the course web page; the module is enclosed in parentheses to facilitate the use of the Church-Rosser Checker and Coherence Checker tools, which do not use built-ins like BOOL. For that reason, the automatic inclusion of BOOL in the module is also disabled; of course, you can interact with the module at the Core Maude level for other purposes just by removing the parentheses):

```
(set include BOOL off .)

(mod UNORDERED-CHANNEL is
   sorts Nat NatList Msg Conf State .
    subsort Msg < Conf .
```

```
  op 0 : -> Nat [ctor] .
  op s : Nat -> Nat [ctor] .
  op nil : -> NatList [ctor] .
  op _;_ : Nat NatList -> NatList [ctor] .   *** list constructor
  op _@_ : NatList NatList -> NatList .      *** list append
  op `[_`,_`] : Nat Nat -> Msg [ctor] .
  op ack : Nat -> Msg [ctor] .
  op null : -> Conf [ctor] .
  op __ : Conf Conf -> Conf [ctor assoc comm id: null] .
  op `{_`,_|_|_`,_`} : NatList Nat Conf NatList Nat -> State [ctor] .

  vars N M J K : Nat .
  vars L P Q : NatList .
  var C : Conf .

  eq nil @ L = L .
  eq (N ; L) @ P = N ; (L @ P) .

  rl [snd] : {N ; L, M | C | P, K} => {N ; L, M | [N, M] C | P, K} .
  rl [rec] :  {L, M | [N, J] C | P, J}  => {L, M | ack(J) C | P @ (N ; nil), s(J)}  .
  rl [rec-ack] : {N ; L, J | ack(J) C | P, M}  => {L, s(J) | C | P, M}  .
endm)
```

That is, the contents of the unordered channel is of course modeled as a *multiset* of messages of sort `Conf`. The entire system state, involving the sender, the channel, and the receiver is a 5-tuple of sort `State`, where the components are:

- a buffer for the sender containing the current list of items to be sent
- a counter for the sender keeping track of the sequence number for items to be sent
- the contents of the unordered channel
- a buffer for the receiver storing the sequence of items already received, and
- a counter for the receiver keeping track of the sequence number for items received.

One essential property of this protocol is of course that it achieves *in-order communication* in spite of the unordered communication medium. The first thing you are asked to do, to gauge your understanding of fundamental concepts in the course, is to specify this in-order communication property as an *invariant*, specifying such an invariant in Maude. Then you will be asked to *verify* such an invariant using Maude. We will always assume that all initial states are of the form:

```
{n1 ; ... ; nk , 0 | null | nil , 0}
```

That is, the sender's buffer contains a list of numbers `n1 ; ... ; nk` and has the counter set to `0`, the channel is empty, and the receiver's buffer is also empty. Also, the receiver's counter is initially set to `0`.

Hints. In specifying the invariant, the auxiliary notion of a list *prefix* may be useful to you, where given lists $L = a_1; \ldots; a_n$ and $L' = b_1; \ldots; b_m$, we say that L is a *prefix* of L' iff either: (1) $L = L'$, or (2) $m > n$, and for $1 \leq i \leq n$ we have $a_i = b_i$. Note also that the definition of the invariant has to be *parametric* on the actual list of items $b_1; \ldots; b_m$ sent by the sender. You should therefore aim at specifying the invariant in a parametric fashion (that is, having such a list as an extra argument) so that when checking the invariant for specific initial states you can just specialize the parametric invariant definition to the sequence of data being sent in that initial state. That is, a normal invariant, say `I`, is typically specified as a predicate

```
op I : State -> Bool .
```

Instead, an invariant parametric on, say, lists of natural numbers, would be a predicate:

```
op I : State NatList -> Bool .
```

where to avoid any conflict with the built-in module `BOOL` in using tools such as the Church-Rosser Checker or the Coherence Checker, instead of importing the `BOOL` module you should either import the module `TRUTH-VALUE` from the Maude prelude (which does not contain any built-in operators), or define your own sort of truth values, say, `MyBool`.

The problem with this simple example is that you cannot verify the invariant using the `search` command in Maude, because, due to the `snd` rule, the number of messages that can be present in the channel is unbounded, so that there is an infinite number of reachable states. You should therefore use an *abstraction*. There are of course several key properties that your abstraction should satisfy:

- the set of states reachable from any initial state should be finite
- the equational theory should be confluent and terminating
- the rules should be coherent with the equations
- the abstraction should preserve the invariant

Due to the presence of associativity, commutativity and identity of multiset union for unordered bags of messages inside the channel, older Maude tools like the Church-Rosser Checker and the Coherence Checker cannot be used to verify some of these properties. However, there is a *new* version of these two tools available with documentation in:

```
http://maude.lcc.uma.es/CRChC/
```

which does support associativity, commutativity, and identity axioms. You should use those new tools, and the latest alpha versions of Maude and of Full Maude. All these tools and the lastest Maude and Full Maude versions can be downloaded directly from the course web page; so you only need to access directly the above tool web page for purposes of consulting these tools' documentation.

Warning. I have repeatedly emphasized in lectures that any analysis using an abstraction is *worthless* and *misleading* if the rules are not ground coherent with the equations. Therefore, in order to avoid that your solution to the exercise becomes itself worthless and misleading (which would be a real pity and shame), I expect you to check with the latest Church-Rossert Checker that the equations you add leave the module locally confluent; and that you use the newest Coherence Checker to verify that your abstraction is indeed ground coherent. The command for checking ground coherence is:

```
(check ground coherence .)
```

The only check that I will not ask of you is a check for the termination of your equations. In principle this should also be done, and you are welcome to do it, but given the length of the exam I will not require that. One last point about this warning is that the *preservation of the invariant* by the abstraction is also a crucial issue, whose failure would render the abstraction worthless; so I will expect from you a precise explanation of why your abstraction preserves the invariant.

Additional Hint. If in your analysis of ground coherence for your abstraction it turns out that some rules are not ground coherent with some given equations, this should not block you from finding a good abstraction; because you can make your abstraction ground coherent by *adding* new rewrite rules that fill in the corresponding diagrams for the given critical pairs. Of course, you also should convince yourself and me that any new critical pairs generated by the addition of such new rules can all be filled in, that is, that the new specification after adding those new rules is ground coherent.

Once you have defined your abstraction, you should *verify* the invariant for the following initial states using the `search` comand in Maude:

```
{0 ; s(0) ; s(s(0)) , 0 | null | nil , 0}
{0 ; s(0) ; s(s(0)) ; s(s(s(0))) , 0 | null | nil , 0}
{0 ; s(0) ; s(s(0)) ; s(s(s(0))) ; s(s(s(s(0)))) , 0 | null | nil , 0}
```

4. The ThreadGame program below:

```
class ThreadGame
{
        public static void main(String[] args)
        {
                (new Process(1)).start() ;
                (new Process(1)).start() ;
        }
}

class Process extends Thread
{
        static int c;

        public Process(int i)
                {
                        c = i ;
                }


        public void run()
        {
                while (true) {
                        c = c + c ;
                }
        }
}
```

is a simple multithreaded Java program which shows the possible data races between two threads accessing a common variable (you saw a variant of this in Lecture 27). Each thread reads the value of the static variable c twice and writes the sum of the two values back to c. The question is what values can c possibly hold during the infinite execution of the program. Theoretically, it has been proved that all natural numbers can be achieved. Here we want to use JavaFAN to verify this conclusion for a specific natural number using the search command. More specifically, for the natural number 99, please write a search command to find a possible execution during which the value of c becomes 99 at some moment. **Hint**: the deadlock search function provided by JavaFAN gives a good starting point and the property translation shows the way to extract the value of a static field from the JavaState. Remember how in the last lecture on JavaFAN it was explained how you can see the Maude command corresponding to invoking a JavaFAN search command for deadlocks. You can then modify this Maude command to obtain the search command you need.

5. The Pipeline Java program below

```
class Main {
static public void main (String argv[]) {
Connector c1, c2, c3;
c1 = new Connector();
c2 = new Connector();
c3 = new Connector();
(new Stage(1, c1, c2)).start();
(new Stage(2, c2, c3)).start();
(new Listener(c3)).start();
for (int i=1; i<2; i=i+1) c1.add(i);
c1.stop();
}
}

class Connector {
```

```
public int queue = -1;
public synchronized int take(){
int value;
while ( queue < 0 )
try {wait();} catch (InterruptedException ex) {}
value = queue; queue = -1; return value;
}
public synchronized void add(int o) { queue = o; notifyAll(); }
public synchronized void stop(){ queue = 0; notifyAll(); }
}

class Stage extends Thread {
int id; Connector c1, c2;
int stop = -1;
public Stage(int i, Connector a1, Connector a2)
{ id = i; c1 = a1; c2 = a2; }
public void run() {
int tmp = -1;
while (tmp != 0)
if ((tmp=c1.take()) != 0){
c2.add(tmp+1);
}
stop = 0;
c2.stop();
}
}

class Listener extends Thread {
Connector c;
int stop = -1;
public Listener(Connector con) { this.c = con; }
public void run(){
int tmp = -1;
while (tmp != 0)
if ((tmp=c.take()) != 0);
stop = 0;
System.out.println("Listener stop.");
}
}
```

simulates a pipeline architecture. A desired property for this program is the propagation of termination: if the first stage stops, the final listener should stop eventually. Please use JavaFAN to verify the correctness of this program w.r.t. the propagation of termination property. **Hint**: a special field, stop, is added to the program to indicate the stoppage of the stages.

Note. The JavaRL website is at `http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java`. The tool should be available for download from there. If you have difficulty downloading it or other reasonable questions regarding the use JavaRL, you can send email to Ralf Sasse (`rsasse@cs.uiuc.edu`).

6. (Rate-monotonic scheduling in Real-Time Maude). Consider a set of periodic tasks t_i, each with a nonzero natural number period p_i and a nonzero natural number execution time e_i. That is, in each of its periods, a task must execute for time e_i. Assume also that there is only one processor in the system to execute the tasks.

   For example, if a task has period 5 and execution time 2, then it must execute for 2 time units during the time interval $[0, 5]$, must execute for two time units during the time interval $[5, 10]$, and so on.

   In contrast to, e.g., EDF scheduling, in *Rate-Monotonic Scheduling* (RMS), the priorities of the tasks are based only on their periods, where a task with a shorter period has a higher priority than a task with longer period. For example, a task with period 5 always has higher priority than a task with period 7.

We assume that tasks can be preempted (i.e., their executions can be paused "in the middle" and be resumed later). It is always the runnable task with the highest priority that must be executed, if necessary by preempting executing tasks with lower priority.

In this exercise, you should define an object-oriented Real-Time Maude model of rate-monotonic scheduling that will work for any given set of tasks; that is, a model that simulates the executions of a set of tasks under the rate-monotonic scheduling policy.

You should explain how one can use Real-Time Maude's unbounded search command (`utsearch`) to check whether a given task set is schedulable (that is, whether each task t_i can always execute for time e_i in each of its rounds).

Given the following task sets:

(a) t_1 has execution time 1 and period 3; and t_2 has execution time 3 and period 5.

(b) t_1 has execution time 2 and period 5; and t_2 has execution time 4 and period 7.

(c) t_1 has execution time 1 and period 6; t_2 has execution time 2 and period 6; and t_3 has execution time 3 and period 7.

(d) t_1 has execution time 1 and period 6; t_2 has execution time 2 and period 6; and t_3 has execution time 4 and period 9.

Use your model of RMS and Real-Time Maude's `utsearch` command to check which of the above task sets (if any) are schedulable (in a single-processor system).

For any of the above task sets which is *not* schedulable under rate-monotonic scheduling, provide a behavior that leads to a missed deadline. **Hint:** Real-Time Maude's search commands cannot at the moment output execution paths, so, if you know that a set of tasks is not schedulable, you might want to then use an LTL model checking command to generate a counterexample behavior.